Connection Networks for Output-driven List Multiprocessing

by

Steven D. Johnson
Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 114

CONNECTION NETWORKS FOR
OUTPUT-DRIVEN LIST MULTIPROCESSING

STEVEN D. JOHNSON

OCTOBER, 1981

Research reported herein was supported (in part) by the National Science Foundation under grant number MCS77-22325.

Connection Networks for Output-driven List Multiprocessing**

Steven D. Johnson*

Abstract

Processor-memory communication is discussed for a multiprocessor whose control is based on call-by-need. Concurrent global memory accesses are arbitrated by a connection network whose design is tailored for list processing. One refinement, called the NEW-sink, reduces the waiting time for a common kind of transaction, the allocation of new list cells. A specific connection scheme, a banyan network, has been simulated to show the effects of design refinements.

CR Categories: 4.32, 6.21.

Keywords: Connection Network, Multiprocessing, call-by-need, list processing.

*Mailing address: Indiana University Computer Science Dept., 101 Lindley Hall, Bloomington, Indiana, 47401.

**Research reported herein was supported (in part) by the National Science Foundation under grant number MCS77-22325.

1.0 INTRODUCTION - A MULTIPROCESS MODEL

The term output-driven refers to a paradigm of computation in which results are produced only when it is determined that they are needed. When confined to the relationship between a function and its arguments, this is known as call-by-need [23]. The only "control structure" is the dispersal of need from its ultimate source, the printer(s) -- which absolutely must produce answers [11]. Of course, these answers are determined by the text of a program. This approach to computation has lead us to view the "data space" as a medium of manifest objects in which are embedded suspensions -- transparent processing entities which converge to needed values.

This heterogeneous relationship between process and data is the foundation for the design of a general purpose list multiprocessor. Most mechanisms for introducing concurrency in programs, especially applicative programs, can be described operationally in terms of suspending constructors. Techniques ranging from collateral argument evaluation [9] to nondeterminisic computations and explicit multiprocessing operators [8,12] have been so described.

Our ultimate goal is to specify the architecture of an extensible computer whose processing resources are transparent to the software being run. "Software independence" is achieved by giving the host system sole responsibility for allocating processors to suspensions. By "computation" we mean the reduction of a functional expression to normal form. More specifically, we adopt the graph reduction techniques found in Pure LISP [20] for computing results. Processors (also called evaluators) traverse lists, some of which are interpreted as programs. These lists reside in a common store and may contain suspensions. Once a suspension has been accessed, some evaluator computes its value.

The overall view, depicted in Figure 1, is that of a self-activating data space. In any foreseeable implementation technology, the number of stored data objects exceeds the number of processing components by several orders of magnitude; there may be far more suspensions in the data space than evaluators. This report focuses on the problem of sharing the store. The issue of control, which evaluator computes which suspension, is not considered here. Access to the store consists of list processing primitives. Messages have a fixed format about the size of a memory cell; since the intended host language is applicative, unconstrained side effects do not occur [10,14].

The global architecture for the system discussed here is shown in Figure 2. A connection network arbitrates concurrent access to the store by some number of processors. The crucial issue is whether a network can be devised in which processing power can grow without bound. Numerous connection networks have been proposed for multiprocessing. The familiar extremes, the bus architecture and the full crossbar switch exemplify the fundamental tradeoff between extensibility and complexity in communication networks. The finite bandwidth of any bus imposes an upper limit on the number of components that can be attached to it. The polynomial hardware cost of a crossbar and the algorithmic complexity of its control make it almost impossible to extend. Between the extremes are innumerable "compromise" architectures. A routing network based on a banyan graph structure [22] has been chosen for discussion in this report. It is moderately extensible and has a highly decentralized control algorithm. Within reasonable bounds its hardware cost is acceptable [6], and it is often put forward as a candidate for general purpose multiprocessing [4, 5,15,17].

Although we concentrate on the role of the connection network in the overall design, in doing so we are forced to characterize the behavior of the other components. Assertions about how processors and memories behave are based on other studies [2,16,18], and/are used to parameterize a simulation model, documented below. The simulation is part of a more ambitious design effort [19], but in this report it serves primarily as a tool for isolating the qualitative effects of enhancements to the design.

In summary, what follows is a cursory description of communication hardware for the management of data space access in an output-driven, applicative machine. It is also a forum for the discussion of the computational components of such a machine. The machine's many processors are geared toward list processing. Section 2 establishes some vocabulary for the rest of the report and describes the local behavior of system components. Section 3 describes the architecture of banyan networks. Section 4 discusses the expected global performance of the proposed machine, followed in Section 5 by a description of the observed performance in simulation. Section 6 is a summary.

2.0 DEFINITIONS AND OVERVIEW.

Although "communication" is the main subject of this report, that term is used in a restricted sense, meaning "the exchange of digital information among a number of components." "Component" too is a base term, meaning "some digital computing device," which may be of general or special purpose. At any time, many individual exchanges occur (or are attempted) concurrently.

1. An <u>agent</u> is any component that communicates.
2. A <u>processor</u> is an agent that initiates communication, an "active" or a "computing" component.
3. A <u>memory</u> is an agent that responds to communication, a "passive" or a "storage" component.
4. A <u>switch</u> is an agent that connects processors to memories, a "routing" component.

In the context of this report the terms "component" and "agent" are synonymous. The ad hoc distinction between processors and memories, while a bit misleading, facilitates the discussion. In general, memories are not necessarily passive; storage components can have considerable decision making power [11]. Moreover, it is not clear into which category storage reclamation agents fall.

A system is built by connecting a set of processors to a set of memories with some configuration of switches. Processors send fixed sized messages such as "Tell me the contents of cell 11," that are delivered to the appropriate memory by the switches. Memories carry out these instructions as the messages arrive.

5. A <u>transaction</u> is a communication between two agents (usually a processor and a memory) disregarding the existence of intervening agents (usually switches).

6. A <u>message</u> is an item of transaction, the abstract unit of communication.

7. A <u>packet</u> is the physical representation of a message.

8. A <u>Transfer</u> is the passage of a packet directly from one agent to another.

9. A <u>Transmission</u> is the sequence of transfers associated with a transaction.

We elect a "packet switching" implementation of transactions: switches are empowered to detain packets in order to manage confluent message streams. Allowing the connection network to absorb messages increases throughput, and hence gross system performance, provided processors can handle more than one transaction at a time. This is the case as long as messages from a given source to a given destination arrive in the order they were issued; the connection network must have the "virtual circuit" property [3].

A transaction is a single access to the store, and a packet is assumed to hold exactly one message. It is wrong to say "a processor transmits a message;" transmission is a physical act, and messages are abstract entities. To maintain a distinction between objects and their implementations, we shall say that messages are sent and received, whereas packets are issued and absorbed.

Computation in the target system is based on the traversal and interpretation of list structures. A processor's observable behavior is a sequence of accesses to the data space. These transactions fall into three categories, analogous to LISP operations:

10. RSVPs (CAR and CDR in LISP) - The sender is "fetching" the contents of a cell. The receiving memory responds by transmitting an answer packet to the processor that issued the request.

11. STINGs (RPLACA and RPLACD in LISP) - are "store" operations. The content of the receiving memory is updated; no acknowledgment is required.

12. NEWs (CONS in LISP) - A processor sends a request for the address of an unreferenced cell. A memory responds by allocating a cell and sending its address to the processor. (Presumably the processor STINGs values into this cell later).

Of concern is the amount of time agents are tied up during transactions. These delays are measured by recording the amount of time agents are in various states of activity.

13. An agent is <u>active</u> when it is sending or receiving messages.

14. An agent is <u>idle</u> if it is involved in work other than communication.

15. An agent is <u>waiting</u> if cannot proceed until a transaction in progress is completed. For example, a processor may have to wait for the response to an RSVP message before it can continue.

16. An agent is <u>blocked</u> if it cannot issue a packet because of the state of the connection network.

"Idle" means "idle with respect to the communication hardware", not that the agent is not computing. Some transmission cost is associated with each kind of transaction, and it depends on several factors. Since RSVPs are two-way transactions and STINGs are not, RSVPs cost more. In all but the most extreme architectures, another factor is the locality of references. The extent to which processors "keep to themselves" largely determines the amount of interference in transmission.

17. <u>Intrinsic delay</u> is the amount of time taken to complete a transaction in the absence of interference.

18. <u>Expected delay</u> is the average transaction time in a "running system", the observed delay in the presence of concurrent transactions.

19. Blockage is the difference between expected and intrinsic delay.

20. Utilization is the percentage of time an agent is active or idle.

Blockage is a symptom of inter-agent interference due to lack of locality, differing agent speeds, communication bottlenecks, etc.. One hundred percent utilization is an impossible goal; even in the presence of a high degree of locality systems fail to achieve intrinsic delay [21, pg. 86].

3.0 BANYAN NETWORKS

Figure 3 depicts a connection network with the structure of a banyan graph [22]. Eight processors (the circles along the bottom of the figure) are connected to eight memories (along the top of the figure) by three stages of four two-by-two crossbar switches. The boxes in the figure, called ports, are the medium of transmission. Each box can hold a small number of packets for transfer by an agent. The switches are not shown as objects; their connectivity is implied by the edges in the figure. The network can be enlarged either by increasing the number of stages or the size of the switches. Only two-by-two switches are considered here. The switches are all electronically identical, as are the ports. Message routing is determined by a fixed, distributed control algorithm, described below. Assume that packets are transmitted in both directions; those issued by processors (travelling upward) are called "outgoing"; those issued by memories are called "incoming".

3.1 Outgoing Message Routing And Source Recovery.

The edges in Figure 3 indicate connections an individual switch can make between ports. Each switch inspects the least significant bit of the destination address of a packet in its input port. If this bit is zero (one) the packet is transferred to the left (right) output port. During the transfer, the destination address is shifted one bit to the right, so that the next bit determines routing at the next stage. Regardless of their source, all packets with the same destination arrive at the same output; no address translation is needed (see Figure 3).

If an RSVP message is sent, the memory must know where to return its answer. This "source" address can be constructed by recording the state of each switch the message packet goes through. As the outgoing decision bit is shifted out of the destination address a new bit is shifted in - a zero (one) if the packet came from the switch's left (right) input port. (It is easier to think of the bit as being <u>replaced</u>.) Since these new bits encode the path taken during transmission, the source address results.

3.2 Address Independence

Its routing capabilities make the banyan network attractive for multiprocessing. Its control algorithm, while simple and highly decentralized, transfers packets with equanimity. Its ability to recover source addresses allows processors to be added, deleted, replaced, or exchanged without keeping track of names. (Alas, the same cannot be said of memories.) It can be built from electronically simple, identical components.

To their detriment, banyan networks cannot be trivially extended. When adding stages there seems to be no way around the increasing number of crossovers of communication paths. In some technologies, VLSI for example, crossovers can be the dominant cost, and the banyan network's asymptotic space complexity is comparable to that of a packet switching crossbar [6]. In addition, such practical considerations as error recovery, which are not dealt with in this report, can add a great deal of complexity to the network control.

But within bounds, say around a million processors, banyans offer an elegantly simple means of message switching, at plausible cost, while imposing an intrinsic delay that varies with the logarithm of the number of processors. Taken alone, logarithmic message delay does not limit the growth of processing power.

4.0 BEHAVIORAL ASSUMPTIONS.

A simulation model must address at least three levels of behavior in the system under study. The highest is the communicative nature of agents. This is modelled by assuming that the processors execute independently and comprise the driving force for the system. Each processor is a stochastic process with four states: IDLE, RSVP, STING, and NEW. Switches and memories respond deterministically to the presence of packets in their ports.

The next level of behavior is the relative performance of the agents. Here the switch cycle is taken as the basic unit of time. This is a consequence of an assumption that agents are implemented in VLSI, one to a chip. Packets are dozens of bits long, making it reasonable to conclude that they will be serialized for transfer; switches are pin-bound. Serialization cannot be done at memory cycle speed, and processors, though more complex than switches, can probably be designed to issue packets at the rate that memories can absorb them. So the packet transfer rate dominates, and together with the negligible routing control logic defines a switch cycle.

At the lowest level is the local synchronous behavior of transmission. Of concern are conditions which lead to blockage and the routing heuristics used to avoid them. Section 6 treats these in more detail.

When a processor issues an RSVP message it waits for a response before changing state*. Intrinsic delay for RSVPs is the number of switches an outgoing RSVP packet goes through to get to the absorbing memory, plus the number of switches the incoming response packet goes through to get back to the source processor. The sum is 2logN in the banyan network**.

Similarly, STING transactions have logN intrinsic delay, However, having issued a STING, a processor need not wait for an acknowledgment. So utilization can approach one hundred percent where processors are only STINGing.

Since NEWs are two-way transactions, they appear to have the character and intrinsic delay of RSVPs. But an enhancement of the switching function makes this form of communication look more like a STING. Each port is given enough capacity to hold the address of one newly allocated

----

*"Lookahead" schemes to reduce waiting are not precluded, but they are beyond the scope of this report. In operational terms, a processor that has to wait W cycles for every fetch can probably handle W suspensions at a time.

**N is the number of processors. The fan-out of the switches, in this case two, determines the base of the logarithm.

cell. When conditions permit, new cells are acquired by the switches and transmitted in the direction of the processors. The connection network maintains a reservoir of anticipated NEW requests acting in a manner analogous to a capacitor or a heat sink. Sporadic NEWs are absorbed by the lower switch stages with little or no waiting.

One goal of the simulation is to study the effect of this enhancement, called the NEW-sink, on locality. The switch should transmit NEW packets (say) directly across the network whenever there is a choice, routing new cells to processors from memories in the same relative position. However, if a processor issues a "burst" of NEW requests, or inversely if a memory has run out of unreferenced cells, regions of the NEW-sink are drained. To accomodate the demand for new cells switches may route them to non-local destinations.

5.0 OBSERVED BEHAVIOR IN THE SIMULATION MODEL

Coding for the simulation is provided in Appendix A. Output from the program was in tabular form, and has been graphed by hand in the accompanying figures.

5.1 The NEW-sink

In the NEW-sink a switch examines its processor-side ports in every cycle. If one or both are empty, the switch tries to transfer NEW packets from its memory-side ports. As noted above, preference is given to transfers straight across the network.

Suppose every processor tries to retrieve a NEW cell in every cycle. Their ports are emptied simultaneously. On the next cycle switches in the first rank discover the empty ports and latch NEW packets into them; the processors are waiting. The ranks of empty ports ripple across the connection network as more highly positioned switches provide new cells. Expected delay is one, and process utilization is fifty percent.

Figures 4 and 5 show utilization and locality under varying processor and memory behaviors. To isolate NEW-sink performance only NEW packets are issued. Processors are given some probability of requesting NEW packets; memories are given some probability of generating them. The simulation contains sixteen processors and sixteen memories. <u>Utilization does not change in simulations of larger systems.</u>

5.1.1 Utilization (Figure 4) - Assume processor activity is fixed. As memories' responsiveness to NEW requests increases so does utilization, until the system reaches its capacity to produce packets. When processors are one hundred percent active the system levels out at expected delay two. This expected delay is achieved when memory is about 75 percent responsive. As processor activity decreases, stable systems achieve higher utilization. Both phenomena indicate the capacitive nature of the NEW-sink. Anticipated NEW requests supply processors at a rate better than could be expected from memory behavior alone. Evidently, memory is asked more often for NEW packets than is accounted for just by processor demands.

5.1.2 Locality (Figure 5) - The NEW-sink does affect locality. A uniformly applied switch preference associates a unique memory with each processor. Locality is the fraction of NEW packets absorbed by a processor that were issued by its associated memory. If memory responsiveness is fixed, locality improves as processor activity increases. If processor activity is fixed, locality tends to improve with memory responsiveness, except in the case where memory is unresponsive. If NEW packets are sparse in the network, they flow directly accross it; although utilization is low, locality is relatively high.

Processor contention for NEW packets is usually resolved in the first stage of the switching network (the lowest stage in Figure 3). Thus almost all the non-local packets a processor absorbs come from a single secondary memory.

5.2 RSVPs (Figure 6)

Where processors issue only RSVP messages expected delay varies negligibly from intrinsic delay, even in the absence of locality. Utilization is 1/2logN for each processor, making "gross utilization" (the amount of work the system is doing) N/2logN. If processors always wait for the response to their RSVP messages, blockage is negligible. Transactions do not interfere with each other because the network is almost empty.

5.3 STINGS (Figure 7)

An attempt to model system behavior where processors send only STING messages underscores the value of simulation as a design aid. Figure 7 shows the performance of processors that STING on every cycle, but with varying degrees of locality. With 100% locality expected delay equals intrinsic delay and utilization is 50%, as should be expected. What was unexpected was that utilization did not approach this limit as locality improved. In fact, the system performed worse when half of all references were local than when only a quarter were.

The fault lay in the switches -- they were poorly designed! In the NEW-sink preference is given to transfers straight across the network (Section 5). In this simulation, the same heuristic is used for outgoing STINGs. As a result, packets that travel vertically block packets that travel diagonally; the network becomes congested. Unless there are enough "holes" in the transmission medium (due to RSVP transactions -- Section 6.2) to alleviate this problem, the switch control algorithm must record enough history to alter its preference from time to time.

5.4 Mixed Transactions (Figure 8)

Figure 8 shows the result of a series of simulations in which processors have the state transition matrix

$$\begin{array}{lcccc} & \text{IDLE} & \text{NEW} & \text{STING} & \text{RSVP} \\ \text{IDLE} & 0 & 1-r & 0 & r \\ \text{NEW} & 0 & 0 & 1 & 0 \\ \text{STING} & 0 & (1-r)/2 & (1-r)/2 & r \\ \text{RSVP} & 0 & (1-r)/2 & (1-r)/2 & r \end{array}$$

The form of this matrix comes from the following assumptions:

1. A processor is never IDLE. It is capable of issuing a memory request on every cycle.
2. Each NEW is immediately followed by a STING. Processors do not "hoard" cells for some private purpose such as buffering.
3. A processor is about as likely to STING a NEW cell as an old one.
4. Since computation is driven by list traversal, RSVPs are the most frequently occurring message.

The steady state vector for this stochastic process is

$$[0,\ (1-r)/(3-r),\ 2(1-r)/(3-r),\ 2r/(3-r)]$$

Figure 8 shows utility under various values for r.

Of these assumptions the third is the most speculative and optimistic. In purely output-driven systems the only side effect allowed is the convergence of suspensions to values [10]. The only cells that can be re-STING'd are those that are uniquely referenced by a process. A modelling effort is underway to determine, among other things, the proportion of uniquely referenced cells in the data space. However, at this time there exists no empirical evidence to support assumption 3. The figure indicates that utility has logarithmic decline as in Figure 7, but is higher than in the pure RSVP system since STINGs and NEWs cause little waiting. Under the assumptions of the simulation, with an RSVP-factor of r = .7, a system of thirty-two processors and memories performs with about eight times the power of one processor.

6.0 CONCLUSIONS AND DIRECTIONS

This report is part of a design effort for a computer based on the relatively high level principle of output-driven computation. A multiprocessing host architecture can exploit the existence of suspensions, primitive processing entities embedded in the data space. The evaluators that activate some of these suspensions must compete for access to a global store, and some aspects of this contention are investigated here.

Computation takes the form of list processing, and the microscopic behavior of agents is characterized by three transactions called STINGs, RSVPs, and NEWs. The focus of this report is the connection network which arbitrates these transactions, and a close look is taken, through simulation, at banyan networks as an example. This network can be assembled from simple, identical components. Its routing capabilities, together with the fact that suspensions are transparent in the data space, makes it possible to vary the number of processors independent of software. The simulation gives qualitative evidence that the banyan network can deliver increased gross utilization as processors are added.

Utilization is dominated by the most frequent (and most costly) transaction, the RSVP. STINGs can be issued with no waiting since a response is not required. By allowing the connection network to anticipate requests for new cells, and to buffer them in the NEW-sink, NEW transactions are served in unit time. The "holes" left in the network by agents waiting for RSVPs gives switches time to acquire NEW packets and to transmit them to the preferred destination.

The merits of message switching must be weighed against its faults; in the shadow of lowered performance expectations, simpler networks such as the bus are competative, even in systems with hundreds of agents. Attempts to increase throughput lead inevitably to more blockage. As the holes disappear in the connection network the NEW-sink becomes less effective and locality suffers.

More quantitative conclusions await refinement in the output-driven model of computation. Processors probably have more complicated behavior than the simple stochastic model presented here. We have assumed that in the worst case, access to the store is random. It may be however, that processors frequently access the same memory locations. Of particular importance is the way storage reclamation fits into this processing scheme.

References

[1] Birtwistle,G.M., Dahl,O. Myhrhaug,B. and Nygaard,K, SIMULA begin, Auerback Publishers Inc., Philadelphia, Pennsylvania, 1973.

[2] Clark,D.W., Measurements of dynamic list structure use in Lisp., IEEE Transactons on Software Engineering SE-5(1):51-59 (January 1979).

[3] Cypser,R.J., Communications Architecture for Distributed Systems, Addison-Wesley, 1973, 711 pages.

[4] Dennis,J.B., Boughton,G.A. and Leung,C.K.C., Building blocks for data flow prototypes., Proc. 7th Annual Symposium on Computer Architecture (May 1980), pp.1-8..

[5] Dias,D.M. and Jump,J.R., Analysis and simulation of buffered delta networks., IEEE Transactions on Computers, C-30(4):273-282 (April 1981).

[6] Franklin,M.A., VLSI performance comparison of banyan and crossbar communications networks., IEEE Transactions on Computers, C-30(4):283-291 (April 1981).

[7] Friedman,D.P. and Wise,D.S., Applicative Multiprogramming, Indiana University Technical Report No. 72 (1979).

[8] Friedman,D.P. and Wise,D.S., An approach to fair applicative multiprogramming., in Semantics of Concurrent Computation, G. Kahn (ed.), Berlin, Springer (1979), pp.203-226.

[9] Friedman,D.P. and Wise,D.S., Aspects of applicative programming for parallel processing., IEEE Transactions on Computers 27(4):289-296 (April 1978).

[10] Friedman,D.P. and Wise,D.S., CONS should not evaluate its arguments., in Automata, Languages and Programming, S.Michaelson and R.Milner (eds.), Edinburgh, Edinburgh University Press (1976), pp. 257-284.

[11] Friedman,D.P. and Wise,D.S., A conditional, interlock-free store instruction., Indiana University Computer Science Department Technical Report No. 74, Bloomington, Indiana (1978).

[12] Friedman,D.P. and Wise,D.S., An indeterminate constructor for applicative programming., Seventh Annual Symposium on Principles of Programming Languages, (January 1980), pp.243-250.

[13] Friedman,D.P. and Wise,D.S., Output driven interpretation of recursive programs, or writing creates and destroys data structures., Information Processing Letters 5(6):155-160 (December 1976).

[14] Friedman,D.P. and Wise,D.S., Unbounded computational structures., Software - Practice and Experience 8:407-416 (1978).

[15] Grit,D.H. and Page,R.L., Eager beaver evaluation on the R-ary N-cube., Colorado State Univ. Techical Report, (March 1981).

[16] Grit,D.M. and Page,R.L., A multiprocessor model for parallel evaluation of applicative programs., Journ. of Digital Systems 4(2):135-151.

[17] Grit,D.M. and Page,R.L., Performance of a multiprocessor for applicative programs.,International Symposium on Computer Performance Modelling, Measurment, and Evaluation, Seventh IFIP W.G. 7.3, Toronto, Ontario, (1980), pp.181-189.

[18] Johnson,S.D., An interpretive model for a language based on suspended construction., M.S. thesis, Indiana University, Bloomington (1977).

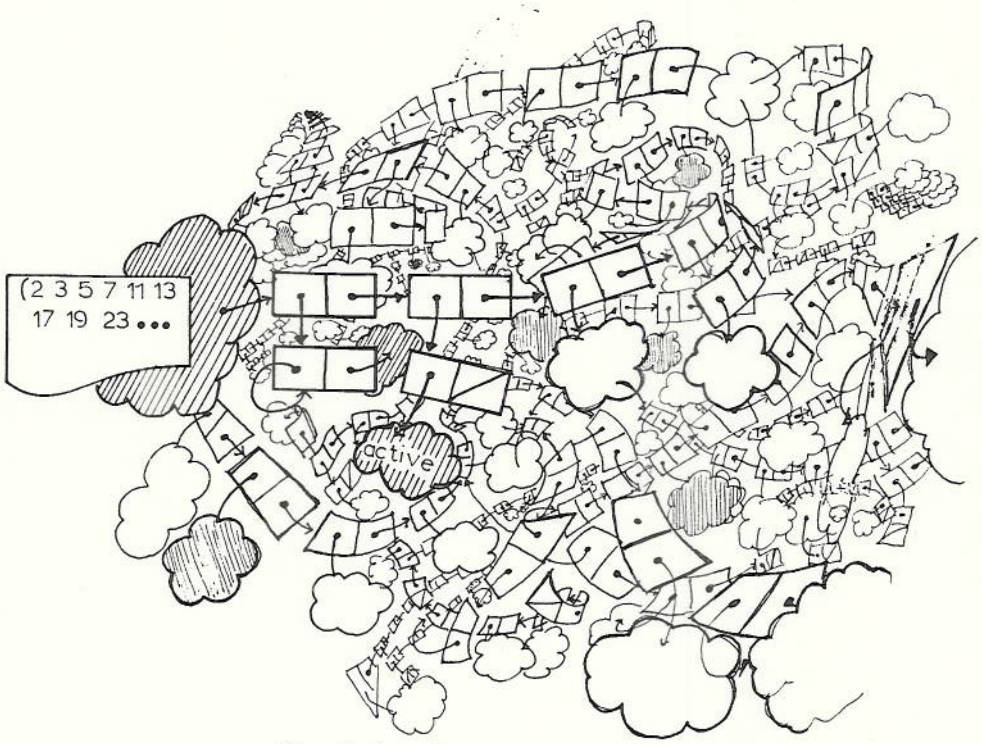
[19] Johnson,S.D., Project description for DSI., in preparation.

[20] McCarthy J, Abrahams,P.W., Edwards,D.J., Hart,T.P. and Levin,M.I., LISP 1.5 Programmer's Manual, The MIT Press, Cambridge, Massachusetts, 1973.

[21] Swan,R.J., The Switching Structure and Addressing Architecture of an Extensible Multiprocessor: CM*, PhD Dissertation, CMU-CS-78-138, Carnegie-Mellonn University, Pittsburgh, Pa., (1978).

[22] Tripathy,A.R. and Lipovski,G.J., Packet switching in banyan networks., Proc. Sixth Annual Symposium on Computer Architecture, (1979), pp.160-167,

[23] Wadsworth,C., Semantics and Pragmatics of Lambda-calculus, Ph.D. Dissertation, Oxford, 1971.



The Data Space

Figure 1

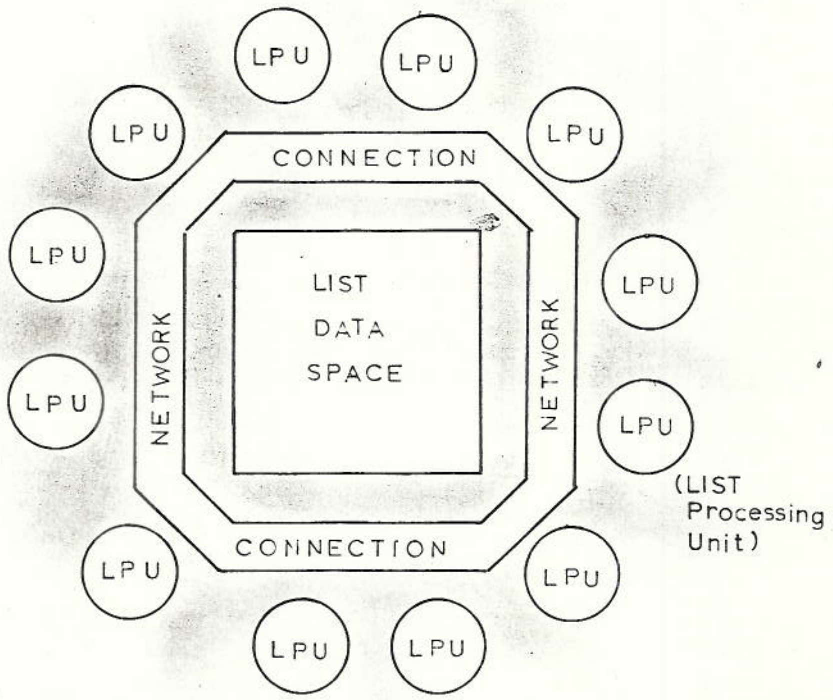


Figure 2

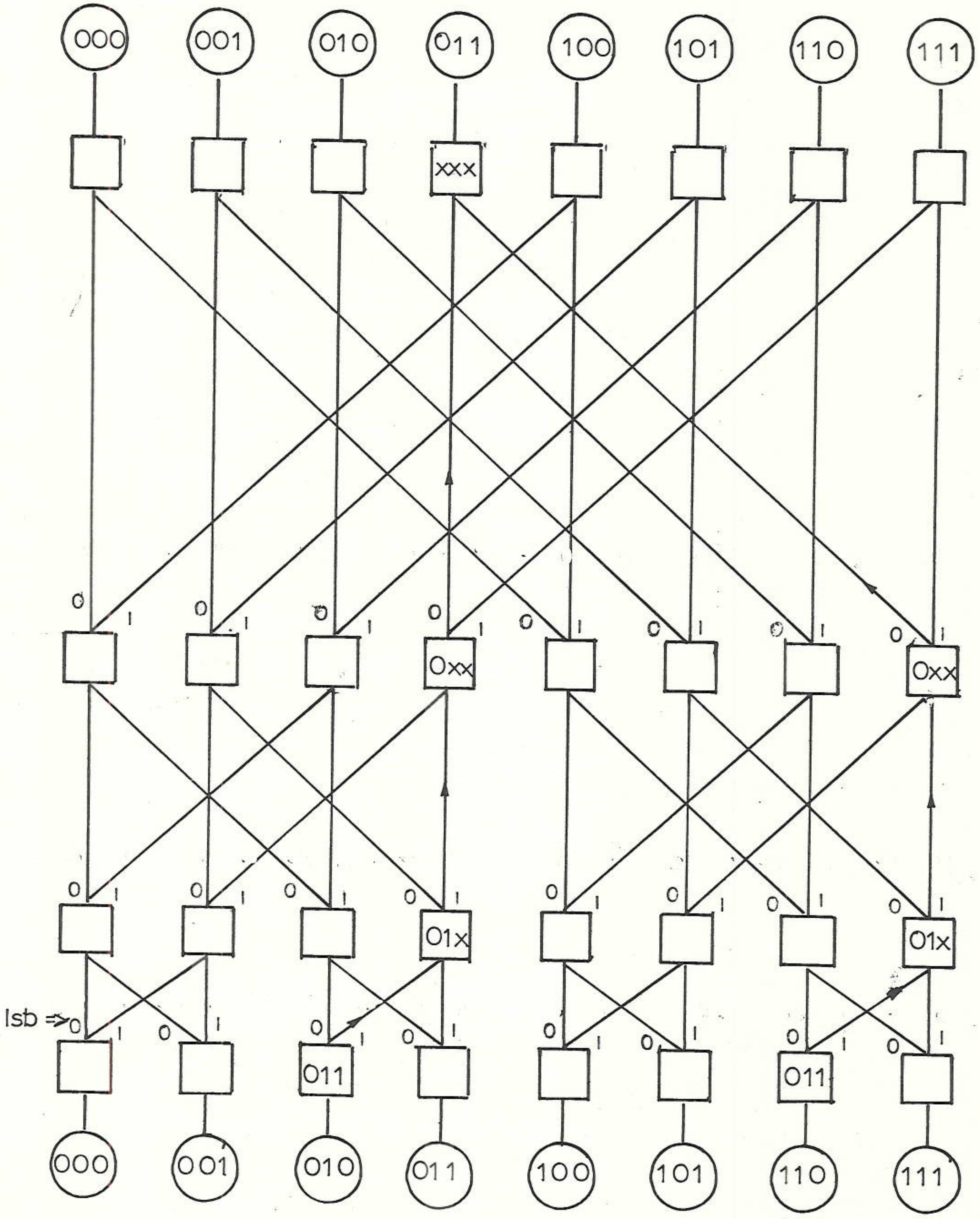


Figure 3 - A three stage network

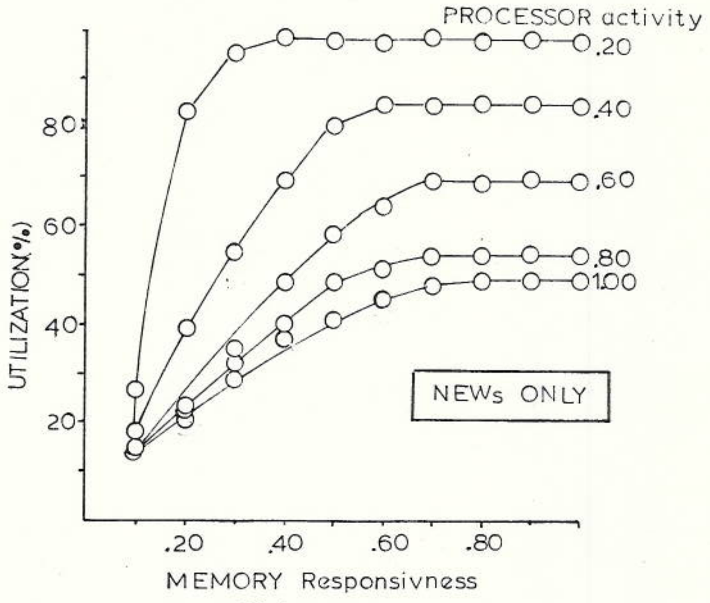


Figure 4

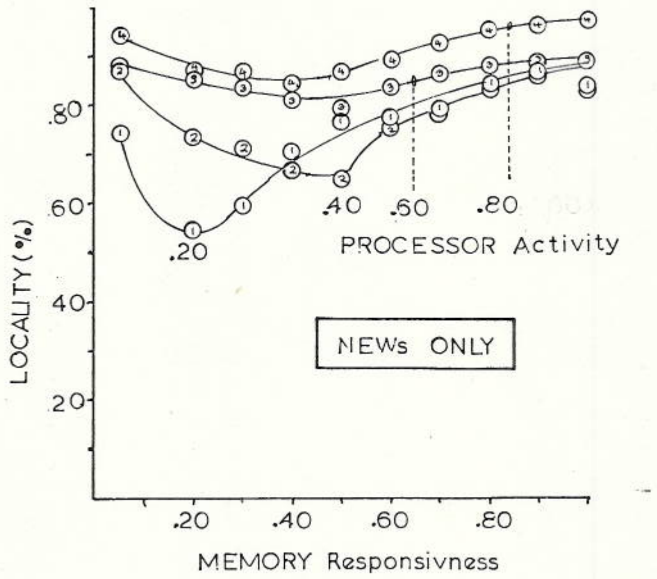


Figure 5

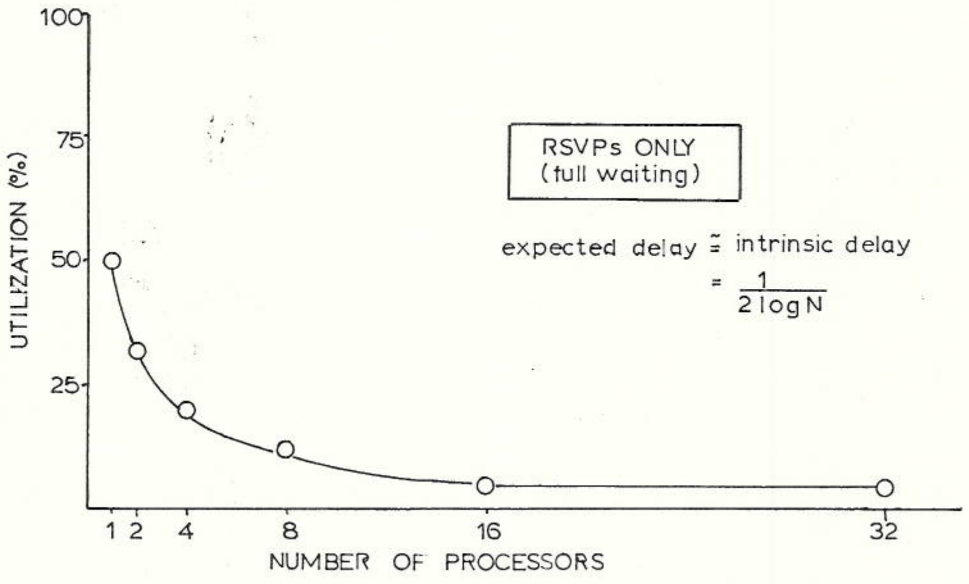


Figure 6

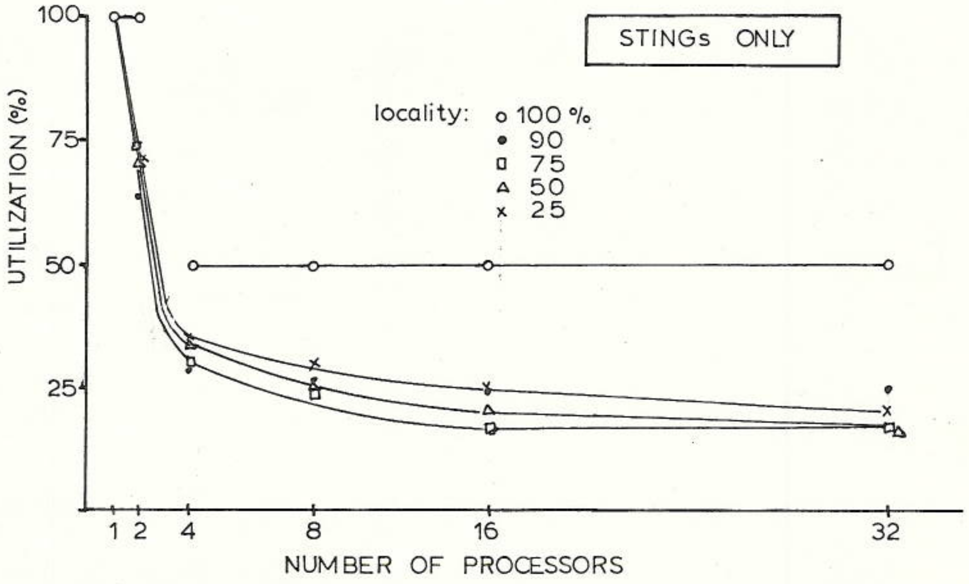


Figure 7

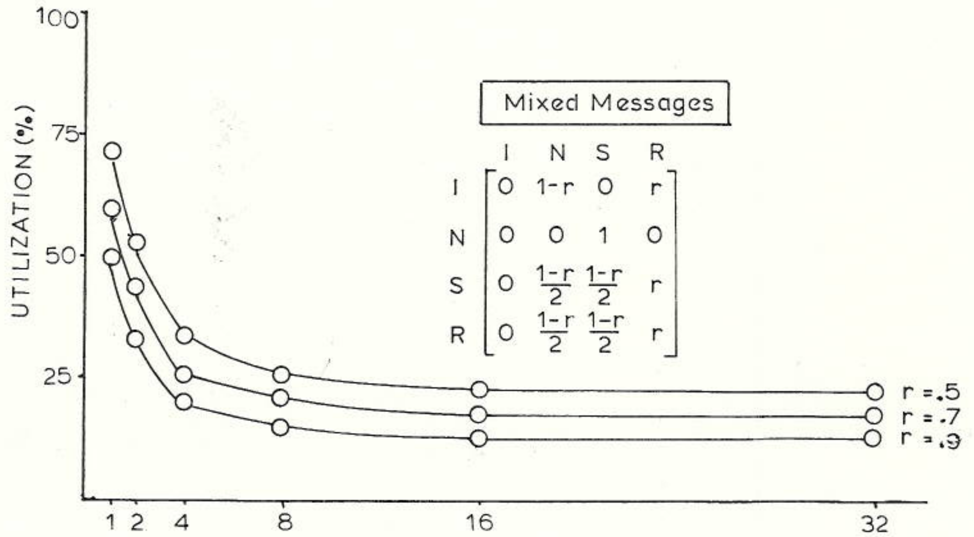

100
75
50
25
UTILIZATION (%)
Mixed Messages
I N S R
I [0 1-r 0 r]
N [0 0 1 0]
S [0 (1-r)/2 (1-r)/2 r]
R [0 (1-r)/2 (1-r)/2 r]
r = .5
r = .7
r = .9
1 2 4 8 16 32
NUMBER OF PROCESSORS

Figure 8

APPENDIX

The Simulation

The connection network model is implemented in SIMULA [1] on a DEC System 10 computer. The packet transmission meduim is a set of PORT objects, able to hold three packets (PKT objects) each, one Outgoing one Incomming and one New. AGENT objects (PROCESSORs, MEMORYs and SWTCHs) have the procedural form:

```
while true do
    begin
    detach;
    {Perform one cycle's worth of local function}
    end
```

Like AGENTs are collected in POOLs (linked lists). In one step of the simulation a POOL resumes each of its members once.

A global transition matrix, TRANS, drives state changes in PROCESSOR objects. By altering the entries in TRANS the relative frequency of transactions and their conditional probabilities are varied.

In each cycle MEMORY objects inspect their PORT. STINGs are absorbed, RSVPs are transfered to the incomming plane unless they are blocked. A NEW packet is generated if it is needed and if no other transfer can take place.

A SWTCH's cycle has two phases. First all SWTCHs inspect their four PORTs, noting the appropriate address bits on the three planes (recall Section 3.0). Global tables LACT and RACT encode the decision procedure for packet transfer. Transfer takes place in the second phase.

```
begin

class PKT(A,RSVP);
   integer A;           comment: Destination address;
   boolean RSVP;        comment: false ==> sting;
   begin
   end PKT;


class PRT;
   begin
   boolean U,L; comment: true ==> (U)pper or (L)ower right port;
   ref(PKT) I;  comment: Latch for Incomming (M --> P) packet;
   ref(PKT) O;  comment: Latch for Outgoing (P --> M) packet;
   ref(PKT) N;  comment: Latch for New-sink (M --> P);
   procedure XFRN(P); ref(PRT) P;
      begin
      N:- P.N; P.N:- none;
      N.A:= (N.A * 2) + (if P.U then 1 else 0);
      end;
   procedure XFRI(P); ref(PRT) P;
      begin
      I:- P.I; P.I:- none;
      if (I.A >= MSB) then I.A:= I.A-MSB;
      I.A := (I.A * 2) + (if P.U then 1 else 0);
      end;
   procedure XFRO(P); ref(PRT) P;
      begin
      O:- P.O; P.O:- none;
      O.A:= (O.A//2) + (if P.L then MSB else 0);
      end;
   end PRT;
```

```
class LIST(ELT); ref(PRT) ELT;
   comment: Used by the routine BANYAN to temporarily tie
        together ports during network construction.  All LIST
        references are discarded before the simulation begins;
   begin
   ref (LIST) REST;
   procedure CONC(L); ref(LIST) L;
      begin
      ref(LIST) T;
      T :- this LIST;
      while (T.REST =/= none) do T :- T.REST;
      T.REST :- L;
      end LIST.CONC;
   REST :- none;
   end LIST;


ref (LIST) procedure BANYAN(N); integer N;
   comment: Construct an N-stage Banyan network of processes,
      interconnected by 2x2 switches.  A list of last stage output
      ports is returned.  The original caller supplies the
      recursively built system with MEMORYs;
   begin
   if (N = 0) then
      begin comment: Base step, build a processor;
      ref(PROCESSOR) P;
      P :- new PROCESSOR(new PRT);
      BANYAN :-new LIST(P.P);
      end
   else
      begin comment: Inductive step, build two subnetworks;
      ref(LIST) BL,BR,T1,T2;
      BL :- BANYAN(N-1);
      BR :- BANYAN(N-1);
      T1 :- BL; T2 :- BR;
      while (T1 =/= none) do
         begin
         ref (SWTCH) S;
         S :- new SWTCH(T1.ELT,T2.ELT);
         T1.ELT :- S.UL; T2.ELT :- S.UR;
         T1 :- T1.REST; T2 :- T2.REST;
         end;
      BL.CONC(BR);
      BANYAN :- BL;
      end;
  end BANYAN;
```

```
class POOL;
   comment: A POOL is a collection of like objects, such as
      PROCESSORs.  It is analogous to a SIMSET.  The
      POOL maintains performance averages and manages the
      resumption of its objects during simulation;
   begin
   ref (AGENT) FIRST,T;
   ref (TALLY) RPT;
   procedure CLOCK(N); integer N;
      comment:  Resume each agent N times. "CLOCK(n+1)" is
         equivalent to "begin CLOCK(1), CLOCK(n) end";
      begin
      while (N>0) do
         begin
         T :- FIRST;
         while (T =/= none) do begin resume(T); T :- T.NEXT; end;
         N := N-1;
         end;
      end POOL.CLOCK;
   procedure RESET;
      begin
      T :- FIRST;
      while (T =/= none) do begin T.RESET; T :- T.NEXT; end;
      end POOL.RESET;
   procedure REPORT(LVL); integer LVL;
      comment:  Get each AGENT to report its performance statistics.
         Place a summary (averages) in this POOL's TALLY.  If the
         report level (LVL) is greater than zero, issue the tally;
      begin
      integer I;
      T:- FIRST; I:= 0;
      while (T =/= none) do
         begin
         RPT.ADD(T.RPT);
         T.REPORT(LVL);
         I:= I+1;
         T:- T.NEXT
         end;
      RPT.AVG(I);
      if (LVL > 0) then
         begin
         outimage;
         inspect FIRST
            when PROCESSOR do outtext("PROCESSOR")
            when MEMORY do outtext("MEMORY")
            when SWTCH do outtext("SWITCH");
         outtext(" pool summary:"); outimage;
         RPT.REPORT(LVL);
         outimage;outimage
         end;
      end REPORT;
   RPT:- new TALLY(new MAP);
   end POOL;
```

```
class AGENT;
   comment: An AGENT is a non-passive message passing object;
   virtual: procedure RESET;
   begin
   ref (AGENT) NEXT;
   ref (TALLY) RPT;
   procedure JOIN(P); ref(POOL) P;
      begin
      this AGENT.NEXT :- P.FIRST;
      P.FIRST:- this AGENT;
      end AGENT.JOIN;
   procedure REPORT(LVL); integer LVL;
      comment:  Issue this AGENT's tally if there is one and if the
         reporting level (LVL) permits;
      begin
      if ((RPT =/= none) and (LVL > 1)) then RPT.REPORT(LVL)
      end AGENT.REPORT;
   inner;
   end;
```

```
AGENΓ class MEMORY(P); ref(PRΓ) P;
   comment:  Withdraw packets from the O-latch of the port.
      Transfer of RSVP messages is blocked if the I-port is
      occupied.  If there is no other transaction, try to
      supply the new-sink;
   begin
   procedure RESET; begin this AGENT.RPΓ:- new TALLY(none) end;
   this AGENT.JOIN(MEMORYPOOL);
   RESET;
   CYCLE:
      detach;
      if (P.O == none) then goto NEWSINK;
      if P.O.RSVP then
         begin
         if (P.I == none) then
            begin
            RPΓ.RA:= RPΓ.RA + 1;
            P.I:- P.O;
            P.O:- none;
            goto CYCLE;
            end
         else if (P.N == none) then
            begin
            RPΓ.RB:= RPΓ.RB+1;
            goto CYCLE;
            end
         else goto NEWSINK;
         end;
      P.O:- none;
      RPΓ.SA:= RPΓ.SA+1;
      goto CYCLE;
   NEWSINK:
      if (P.N == none) then
         begin
         RPΓ.NA:= RPΓ.NA+1;
         P.N:- new PKΓ(0,false);
         end
      else RPΓ.IDL:= RPΓ.IDL+1;
      goto CYCLE;
   end MEMORY;
```

```
AGENT class PROCESSOR(P); ref(PRT) P;
   comment: Generate MEMORY requests;
   begin
   integer STATE, NEXT, ID;
   procedure RESET;
      begin
      STATE:= 0;
      this AGENT.RPT:- new TALLY(new MAP)
      end PROCESSOR.RESET;
   comment:  Process behavior: i) To initialize, enter the processor
      pool, get a REPORT object, then fetch a NEW message - this floods
      the switch network and establishes this processor's ID.
      ii) Throughout the remainder of the processor's lifetime issue
      memory requests as specified in the TRANSition table;
   RESET;
   this AGENT.JOIN(PROCESSPOOL);
   while (P.N == none) do detach;
   RPT.IDL:= this PROCESSOR.ID:= P.N.A;
   P.N:- none;
   CYCLE:
      detach;
   CONTROL:
      NEXT:= randint(0,3,SEED);
      if not draw(TRANS[STATE,NEXT],SEED) then goto CONTROL;
      STATE:= NEXT;
      if (STATE = 1) then goto doNEW;
      if (STATE = 2) then goto doSTING;
      if (STATE = 3) then goto doRSVP;
   doNOOP:
      RPT.IDL:= RPT.IDL+1;
      goto CYCLE;
   doNEW:
      while (P.N == none) do begin RPT.NW:= RPT.NW+1; detach end;
      RPT.NA:= RPT.NA+1;
      RPT.M.INCR(P.N.A);
      P.N:- none;
      goto CYCLE;
   doSTING:
      while (P.O =/= none) do begin RPT.SB:= RPT.SB+1; detach end;
      RPT.SA:= RPT.SA+1;
      P.O:- new PKT(randint(0,MAXADDR,SEED),false);
      goto CYCLE;
   doRSVP:
      while (P.O =/= none) do begin RPT.RB:= RPT.RB+1; detach end;
      if draw(LOCALITY,SEED) then P.O:- new PKT(ID,true) else
      P.O:- new PKT(randint(0,MAXADDR,SEED),true);
      while (P.I == none) do begin RPT.RW:= RPT.RW+1; detach end;
      RPT.RA:= RPT.RA+1;
      P.I:- none;
      goto CYCLE;
   end;
```

```
AGENT class SWTCH(LL,LR); ref(PRT) LL,LR;
   comment:
              This object is a 2x2 crossbar packet switch.  Its cycle
              of behavior has two phases - first to sense the condition
              of its ports, and second to direct those ports to
              transfer packets among each other;
   begin
   ref(PRT) UL,UR;
   integer ISTATUS, OSTATUS, NSTATUS;
   comment:  Initialize the switch.  Mark the ports to indicate
      whether they are leftmost wrt this switch;
   LL.L:= false;
   LR.L:= true;
   UL :- new PRT;  UL.U:= false;
   UR :- new PRT;  UR.U:= true;
   this AGENT.JOIN(SWITCHPOOL);
   while true do
      begin
      detach;
      comment: Phase 1 - Determine the status of the ports;
      B0:= if (LR.O == none) then 0 else CHKLSB(LR.O.A);
      B1:= if (LL.O == none) then 0 else CHKLSB(LL.O.A);
      B2:= if (UR.O == none) then 0 else 1;
      B3:= if (UL.O == none) then 0 else 1;
      OSTATUS:= ((B3*2 + B2)*3 + B1)*3 + B0;
      B0:= if (UR.I == none) then 0 else CHKMSB(UR.I.A);
      B1:= if (UL.I == none) then 0 else CHKMSB(UL.I.A);
      B2:= if (LR.I == none) then 0 else 1;
      B3:= if (LL.I == none) then 0 else 1;
      ISTATUS:= ((B3*2 + B2)*3 + B1)*3 + B0;
      B0:= if (LR.N == none) then 0 else 1;
      B1:= if (LL.N == none) then 0 else 1;
      B2:= if (UR.N == none) then 0 else 1;
      B3:= if (UL.N == none) then 0 else 1;
      NSTATUS:= ((B3*2 + B2)*2 + B1)*2 + B0;
      detach;
      comment: Phase 2 - transfer packets;
      X:= LACT[OSTATUS];
      if (X = -1) then UL.XFRO(LR) else if (X = 1) then UL.XFRO(LL);
      X:= RACT[OSTATUS];
      if (X = -1) then UR.XFRO(LL) else if (X = 1) then UR.XFRO(LR);
      X:= LACT[ISTATUS];
      if (X = -1) then LL.XFRI(UR) else if (X = 1) then LL.XFRI(UL);
      X:= RACT[ISTATUS];
      if (X = -1) then LR.XFRI(UL) else if (X = 1) then LR.XFRI(UR);
      X:= LSNK(NSTATUS);
      if (X = -1) then LL.XFRN(UR) else if (X = 1) then LL.XFRN(UL);
      X:= RSNK(NSTATUS);
      if (X = -1) then LR.XFRN(UL) else if (X = 1) then LR.XFRN(UR);
      end;
  end;
```

```
class TALLY(M); ref(MAP) M;
   comment: This object keeps behavior statistics for its owner.
      Updates, operations and output are handled here;
   begin
   real IDL;      comment: Counts IDLE     cycles;
   real NW,NA;    comment: Counts NEW   Wait and active cycles;
   real SB,SA;    comment: Counts STING  Wait and Active cycles;
   real RB,RA,RW;comment: Counts RSVP   Wait and Active and Busy cycles;
   procedure ADD(T); ref(TALLY) T;
      begin
      IDL:= IDL+T.IDL; NW:= NW+T.NW; NA:= NA+T.NA; SB:= SB+T.SB;
      SA:= SA+T.SA; RB:= RB+T.RB; RA:= RA+T.RA; RW:= RW+T.RW;
      end;
   procedure AVG(X); integer X;
      begin
      IDL:= IDL/X; NW:= NW/X; NA:= NA/X; SB:=SB/X; SA:= SA/X;
      RB:= RB/X; RA:= RA/X; RW:= RW/X;
      end;
   procedure REPORT(LVL); integer LVL;
      begin
      real TA,TW,TB,UT;
      TA:= NA+SA+RA+IDL;  TW:=NW+RW; TB:= SB+RB;
      UT:= if (TA = 0) then 0 else 100*(TA/(TA+TW+TB));
      outtext("   Util: ");outfix(UT,0,3);outtext("% -- ");
      outfix(TA,2,7);outtext("a, ");outfix(TW,2,7);outtext("w, ");
      outfix(TB,2,7);outtext("b, ");outfix(IDL,2,7); outtext(" idle.");
      outimage; outtext("   N,S,R (effort): ");
      UT:= if (NA = 0) then 0 else 100*(NA/(NA+NW));
      outfix(NA,2,7);outtext(" (");outfix(UT,0,3);
      UT:= if (SA = 0) then 0 else 100*(SA/(SB+SA));
      outtext("%), ");outfix(SA,2,7);
      outtext(" (");outfix(UT,0,3); outtext("%), ");
      UT:= if (RA = 0) then 0 else 100*(RA/(RA+RB+RW));
      outfix(RA,2,7);outtext(" (");
      outfix(UT,0,3); outtext("%)."); outimage;
      if (M =/= none) then begin M.RATE; M.REPORT(LVL) end;
      outimage;
      end TALLY.REPORT;
  end TALLY;
```

```
class MAP;
   comment:  An array of counters for measuring locality;
   begin
   real array C(0:MAXADDR);
   real TOT,MX1,MX2;
   procedure INCR(N); integer N; begin C[N]:= C[N]+1 end;
   procedure RATE;
      begin
      integer I,J;
      for I:= 0 step 1 until MAXADDR do TOT:= TOT+C[I];
      if (TOT>0) then for I:= 0 step 1 until MAXADDR do
         begin
         J:= (C[I]/TOT)*100;
         if (J > MX1) then begin MX2:= MX1; MX1:= J end
         else if (J > MX2) then MX2:= J;
         C[I]:= J

         end
      end MAP.RATE;
   procedure REPORT(LVL); integer LVL;
      begin
      if (LVL > 3) then
         begin
         outtext("   Locality -   (1st)  ");outfix(MX1,0,3);
         outtext("%, (2nd) ");outfix(MX2,0,3);outtext("%");outimage;
         end;
      if (LVL > 4) then
         begin
         integer I,J;
         I:= 0;
         REPEAT: outtext("   ");
         for J:= 1 step 1 until 5 do
            begin
            if (I = CAPACITY) then goto TAEPER;
            outint(I,3);outtext(": ");outfix(C[I],0,3);outtext("%     ");
            I:= I+1;
            end;
         outimage;
         goto REPEAT;
         TAEPER: outimage;
         end;
      end MAP.REPORT;
   end MAP;
```

```
integer procedure CHKMSB(A); integer A;
   begin
   CHKMSB:= if (A < MSB) then 1 else 2;
   end;


integer procedure CHKLSB(A); integer A;
   begin
   CHKLSB:= if (A = (A//2)*2) then 1 else 2;
   end;

Comment: Global variables;
   integer SPREAD;         comment: Operator gives number of processors;
   integer CAPACITY;       comment: Number of Proc/Mem pairs;
   integer MAXADDR;         comment: Maximum addressable memory;
   integer MSB;            comment: Leftmost bit in an address;
   integer RUN;            comment: Number of iterations in simulation;
   integer LEVEL;          comment: For reports;
   real LOCALITY,PRRSVP,PRSTNG,PRNEW;
   integer SEED;           comment: For randomizer;
   integer L1,L2,L3;       comment: Loop control;
   integer X,B0,B1,B2,B3;
   integer array LACT(0:35), RACT(0:35), LSNK(0:15), RSNK(0:15);
   real array TRANS(0:3,0:3);
   ref (LIST) L;
   ref (PROCESSOR) P1,P2;
   ref (MEMORY) M;
   ref (POOL) SWITCHPOOL,MEMORYPOOL,PROCESSPOOL;
```

comment: PROCESSOR TRANSITION MATRIX

A matrix of conditional probabilities for processor behavior. States in the stochastic process are IDLE (I), NEW (N), STING (S), and RSVP (R). Rows represent the current state and columns indicate the conditional probability that a PROCESSOR will go into another state.

$$T> \quad \begin{array}{c|cccc|} & I & N & S & R \\ I & 0.0 & 0.0 & 0.0 & 1.0 \\ N & 0.0 & 0.0 & 0.0 & 1.0 \\ S & 0.0 & 0.0 & 0.0 & 1.0 \\ R & 0.0 & 0.0 & 0.0 & 1.0 \end{array}$$

This matrix: PROCESSORs only do RSVPs.

end PROCESSOR TRANSITION MATRIX comment;

comment: SWITCH CONTROL TABLE

In the first phase of switch control the status of ports is checked on each comminication plane. The status is encoded and the second phase switch action is determined by table lookup. A separate table is used for the NEW-sink.

In the incoming/outgoing table:

i) output status codes are 0 (empty) and 1 (occupied)
ii) input status codes are 0 (empty) 1 (go left) and 2 (go right)
iii) transfer actions are ":" (noop), "=" (bar), and "x" (cross)
iv)

plane	L-inp	R-inp	L-out	R-out	R-xfr	L-xfr	bit
outgoing	LL.O	LR.O	UL.O	UR.O	UR	UL	LSB
incoming	UL.I	UR.I	LL.I	LR.I	LR	LL	MSB

	OUT		INP		XFR		Comment
S>	L	R	L	R	L	R	
0	0	0	0	0	:	:	
1	0	0	0	1	x	:	
2	0	0	0	2	:	=	
3	0	0	1	0	=	:	
4	0	0	1	1	=	:	R blocked (switch preference)
5	0	0	1	2	=	=	
6	0	0	2	0	:	x	
7	0	0	2	1	x	x	
8	0	0	2	2	:	=	L blocked (switch preference)
9	0	1	0	0	:	:	
10	0	1	0	1	x	:	
11	0	1	0	2	:	:	R blocked (output locked)
12	0	1	1	0	=	:	
13	0	1	1	1	=	:	R blocked (switch preference)
14	0	1	1	2	=	:	R blocked (output locked)
15	0	1	2	0	:	:	L blocked (output locked)
16	0	1	2	1	x	:	L blocked (output locked)
17	0	1	2	2	:	:	R,L blocked (output locked)
18	1	0	0	0	:	:	
19	1	0	0	1	:	:	R blocked (output locked)
20	1	0	0	2	:	=	
21	1	0	1	0	:	:	L blocked (output locked)
22	1	0	1	1	:	:	R,L blocked (output locked)
23	1	0	1	2	:	=	L blocked (output locked)
24	1	0	2	0	:	x	
25	1	0	2	1	:	x	R blocked (output locked)
26	1	0	2	2	:	=	L blocked (switch preference)
27	1	1	0	0	:	:	Outputs locked
28	1	1	0	1	:	:	" "
29	1	1	0	2	:	:	" "
30	1	1	1	0	:	:	" "
31	1	1	1	1	:	:	" "
32	1	1	1	2	:	:	" "
33	1	1	2	0	:	:	" "
34	1	1	2	1	:	:	" "
35	1	1	2	2	:	:	" "

end SWITCH CONTROL TABLE comment;

comment: NEW-sink CONTROL TABLE

	status				action		comment
N>	UL	UR	LL	LR	LL	LR	
N>							
0	0	0	0	0	:	:	
1	0	0	0	1	:	:	
2	0	0	1	0	:	:	
3	0	0	1	1	:	:	
4	0	1	0	0	:	=	Switch preference
5	0	1	0	1	x	:	
6	0	1	1	0	:	=	
7	0	1	1	1	:	:	Outputs locked
8	1	0	0	0	=	:	Switch preference
9	1	0	0	1	=	:	
10	1	0	1	0	:	x	
11	1	0	1	1	:	:	Outputs locked
12	1	1	0	0	=	=	Switch preference
13	1	1	0	1	=	:	Switch preference
14	1	1	1	0	:	=	Switch preference
15	1	1	1	1	:	:	Outputs locked.

end NEW-sink CONTROL comment;

```
comment: The TRANSition matrix for PROCESSORs declares the
   probabilities that a process will do one operation, given that
   in the last iteration it did another operation.

   0 (Idle)            1 (New)              2 (Sting)          3 (RSVP)       ;
   TRANS[0,0]:= 0   ;TRANS[0,1]:= 0   ; TRANS[0,2]:= 0   ;TRANS[0,3]:= 1   ;
   TRANS[1,0]:= 0   ;TRANS[1,1]:=0 ; TRANS[1,2]:=0 ;TRANS[1,3]:=1 ;
   TRANS[2,0]:= 0   ;TRANS[2,1]:=0 ; TRANS[2,2]:=0 ;TRANS[2,3]:=1 ;
   TRANS[3,0]:= 0   ;TRANS[3,1]:=0 ; TRANS[3,2]:=0 ;TRANS[3,3]:=1 ;
LACT[ 0]:= 0; LACT[ 1]:=-1; LACT[ 2]:= 0; LACT[ 3]:=+1; LACT[ 4]:=+1;
LACT[ 5]:=+1; LACT[ 6]:= 0; LACT[ 7]:=-1; LACT[ 8]:= 0; LACT[ 9]:= 0;
LACT[10]:=-1; LACT[11]:= 0; LACT[12]:=+1; LACT[13]:=+1; LACT[14]:=+1;
LACT[15]:= 0; LACT[16]:=-1; LACT[17]:= 0; LACT[18]:= 0; LACT[19]:= 0;
LACT[20]:= 0; LACT[21]:= 0; LACT[22]:= 0; LACT[23]:= 0; LACT[24]:= 0;
LACT[25]:= 0; LACT[26]:= 0; LACT[27]:= 0; LACT[28]:= 0; LACT[29]:= 0;
LACT[30]:= 0; LACT[31]:= 0; LACT[32]:= 0; LACT[33]:= 0; LACT[34]:= 0;
LACT[35]:= 0;
LSNK[0 ]:= 0; LSNK[1 ]:= 0; LSNK[2 ]:= 0; LSNK[3 ]:= 0; LSNK[4 ]:= 0;
LSNK[5 ]:=-1; LSNK[6 ]:= 0; LSNK[7 ]:= 0; LSNK[8 ]:=+1; LSNK[9 ]:=+1;
LSNK[10]:= 0; LSNK[11]:= 0; LSNK[12]:=+1; LSNK[13]:=+1; LSNK[14]:= 0;
LSNK[15]:= 0;
RACT[ 0]:= 0; RACT[ 1]:= 0; RACT[ 2]:=+1; RACT[ 3]:= 0; RACT[ 4]:= 0;
RACT[ 5]:=+1; RACT[ 6]:=-1; RACT[ 7]:=-1; RACT[ 8]:=+1; RACT[ 9]:= 0;
RACT[10]:= 0; RACT[11]:= 0; RACT[12]:= 0; RACT[13]:= 0; RACT[14]:= 0;
RACT[15]:= 0; RACT[16]:= 0; RACT[17]:= 0; RACT[18]:= 0; RACT[19]:= 0;
RACT[20]:=+1; RACT[21]:= 0; RACT[22]:= 0; RACT[23]:=+1; RACT[24]:=-1;
RACT[25]:=-1; RACT[26]:=+1; RACT[27]:= 0; RACT[28]:= 0; RACT[29]:= 0;
RACT[30]:= 0; RACT[31]:= 0; RACT[32]:= 0; RACT[33]:= 0; RACT[34]:= 0;
RACT[35]:= 0;
RSNK[0 ]:= 0; RSNK[1 ]:= 0; RSNK[2 ]:= 0; RSNK[3 ]:= 0; RSNK[4 ]:=+1;
RSNK[5 ]:= 0; RSNK[6 ]:=+1; RSNK[7 ]:= 0; RSNK[8 ]:= 0; RSNK[9 ]:= 0;
RSNK[10]:=-1; RSNK[11]:= 0; RSNK[12]:=+1; RSNK[13]:= 0; RSNK[14]:= +1;
RSNK[15]:= 0;
```

```
outtext("Banyan simulation.");
outimage;
outtext("Report Level [0-9]: ");breakoutimage; LEVEL:= inint;
outtext("Length of RUN: ");breakoutimage; RUN:= inint;
GETLOC:
outtext("Locality: ");breakoutimage; LOCALITY:= inreal;
outtext("Pr(RSVP): ");breakoutimage; PRRSVP:= inreal;
if (PRRSVP > 1.0) then goto GETLOC;
if (PRRSVP < 0.0) then goto GETLOC;
   TRANS[0,0]:= TRANS[1,0]:= TRANS[2,0]:= TRANS[3,0]:= 0.0;
   TRANS[1,1]:= TRANS[0,2]:= TRANS[1,3]:= 0.0;
 / TRANS[1,2]:= 1.0;
   TRANS[0,3]:= TRANS[2,3]:= TRANS[3,3]:= PRRSVP;
   TRANS[2,1]:= TRANS[2,2]:= TRANS[3,1]:= TRANS[3,2]:= (1.0-PRRSVP)/2.0;
INIT:
outtext("How many stages?: "); breakoutimage; SPREAD := inint;
   CAPACITY := 2.0 ** SPREAD;
   MAXADDR:= CAPACITY-1;
   MSB := 2.0 ** (SPREAD-1);
   SWITCHPOOL :- new POOL;
   MEMORYPOOL :- new POOL;
   PROCESSPOOL :- new POOL;

comment: Build a Banyan network;
L:- BANYAN(SPREAD);
while (L =/= none) do begin M :- new MEMORY(L.ELT); L :- L.REST; end;
outtext("Network constructed."); outimage;outimage;outimage;


for L1:= 1 step 1 until RUN do
   begin
   PROCESSPOOL.CLOCK(1);
   SWITCHPOOL.CLOCK(2);
   MEMORYPOOL.CLOCK(1);
   end;

PROCESSPOOL.REPORT(LEVEL);
comment: MEMORYPOOL.REPORT(1);
outimage; outimage; outimage;
goto INIT;
EXIT:
end.
```